Science of Computer Programming 75 (2010) 1001–1021

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Improving Web Service descriptions for effective service discovery

Juan Manuel Rodriguez, Marco Crasso *, Alejandro Zunino, Marcelo Campo

ISISTAN Research Institute, Universidad Nacional del Centro de la provincia de Buenos Aires (UNICEN), Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina
Consejo Nacional de Investigaciones Cientficas y Tcnicas (CONICET), Argentina

ARTICLE INFO

Article history:
Received 19 May 2009
Received in revised form 21 December 2009
Accepted 3 January 2010
Available online 13 January 2010

Keywords:
Web Services
Web Service publication
Web Service discovery
Web Service discoverability anti-patterns

ABSTRACT

Service-Oriented Computing (SOC) is a new paradigm that replaces the traditional way to develop distributed software with a combination of discovery, engagement and reuse of third-party services. Web Service technologies are currently the most adopted alternative for implementing the SOC paradigm. However, Web Service discovery presents many challenges that, in the end, hinder service reuse. This paper reports frequent practices present in a body of public services that attempt to prevent the discovery of any service. In addition, we have studied how to solve the discoverability problems that these bad practices cause. Accordingly, this paper presents a novel catalog of eight Web Service discoverability anti-patterns. We conducted a comparative analysis of the retrieval effectiveness of three discovery systems by using the original body of Web Services versus their corrected version. This experiment shows that the removal of the identified anti-patterns eases the discovery process by allowing the employed discovery systems to rank more relevant services before non-relevant ones, with the same queries. Moreover, we conducted a survey to collect the opinions from 26 individuals about whether the improved descriptions are more intelligible than the original ones. This experiment provides more evidence of the importance of correcting the observed problems.

1. Introduction

The growing complexity of software systems makes component reuse one of the most valuable tools for software engineers [1]. An evolutionary process currently taking place in the software industry is shifting from developing specific functionality from scratch to discovering and combining functionalities offered by third-parties. Service-oriented computing (SOC) is a new paradigm for building software systems, in which developers look for independent loosely coupled software pieces, called *services*, to form their applications [2–4].

There are three starring players within the SOC paradigm: a service provider, a service consumer and a service discovery system. A service is a unit of work offered by a provider through a publicly available interface. The service discovery system represents a crossroad in the path of providers and consumers. Providers can use the discovery system to advertise their services, while consumers can use it to look for services that match their needs.

Encouraged by the rapid advances in distributed system technologies, the trend in software development has been converging towards reusing and composing services that can be reached using Web technologies. The growth of the Internet has popularized large repositories of software services, called *Web Services* [5]. Web Services are software systems that can be discovered and invoked using standard Web protocols, while each service can still be implemented in a black-box manner. Platform neutrality and self-descriptiveness make Web Services suitable for building networks of applications distributed within and across organizational boundaries.

* Corresponding author.
E-mail address: mcrasso@gmail.com (M. Crasso).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.01.002

Although there is a consensus about Web Service technologies and SOC promise of loose coupling between components, agility to respond to changes in requirements, transparent distributed computing and lower ongoing investments [2], Web Services are currently not as broadly shared and reused as expected [6,7]. One main reason that hinders the adoption of this technology and SOC is that efficient Web Service discovery presents many challenges [8].

The Web Service reference specification for implementing service registries is Universal Description, Discovery and Integration (UDDI).[1] With this standard, publishing services consists of supplying either intra-organization or inter-organization registries with the information associated with providers and technical descriptions of the functionalities of their services in Web Service Description Language (WSDL).[2] WSDL is an XML-based language for describing a service intended functionality using an interface with methods and arguments, in object-oriented terminology, and documentation as textual comments. For example, a provider may describe the interface of a service operation as "getTemperature(zip:string):double." Regarding discovery with UDDI, a discoverer may look for third-party services by sending keyword-based queries to a registry, which returns a list of candidate services in response. Then, the discoverer analyzes the retrieved services and selects the most suitable for his/her needs.

Unfortunately, as the number of published Web Services increases, discovering proper services using the keyword-based support provided by the UDDI standard becomes similar to finding a needle in a haystack [8]. Many problems related to the efficiency of UDDI-compliant approaches to service discovery may stem from the fact that current standards to describe Web Services are incorrectly or not fully employed by publishers in practice [9,10]. Even worse, unlike traditional software libraries, Web Service repositories rely on little meta-data to support discovery [11]. Therefore, if these meta-data do not properly represent the services being published, they will have meagre chances of being discovered.

The problem related to Web Service discovery has been tackled from three main directions. Two of these are difficult to adopt in practice, whereas the other one has shown to effectively ease the human discoverer's tasks while being transparent to adopt. Specifically, one direction proposes to enhance service descriptions instead of exploiting them as they are. The Semantic Web effort proposes to annotate Web Service descriptions using non-ambiguous concept definitions from shared ontologies. By assuming that services are precisely described, it is expected that finding them will be simplified at the expense of increasing development effort [7]. Unfortunately, semantics-based approaches for service discovery suffer from the typical problems associated with ontologies, namely the high complexity involved in building them [12,13], the lack of standard ontologies, and the absence of public semantically annotated Web Services [14].

Another direction proposes Web search engines (e.g., Google) as new sources for finding Web Services [15,16]. As service descriptions usually reside in Web servers, the idea is to exploit the capabilities of Web search engines to crawl and index servers' content. Although this approach is transparent to publishers, several studies have experimentally shown that the precision of Web search engines when looking for known services does not significantly improve, even when proper comments have been introduced in the indexed WSDL documents [15]. Accordingly, Web Service discoverers experience the same problems as ordinary users of Web search engines when trying to adopt this direction.

An interesting direction proposes to adapt classic Information Retrieval (IR) techniques, such as word sense disambiguation, stop-words removal, and stemming for extracting relevant information conveyed within WSDL documents. Obviously, a syntactic approach cannot completely replace the need for semantic descriptions in the context of systems developed via automatic Web Service discovery, in which heterogeneous applications expose and consume services without human intervention [9,17]. However, several IR-inspired approaches [18,19] have been evaluated with different data-sets of nearly 400 Web Services each, showing that they can effectively facilitate human discoverers' tasks without requiring all the specifications of full semantic techniques. Specifically, both [18] and [19] retrieve 78%–71% of the relevant services before retrieving a service non-relevant to the applied queries.

Unfortunately, the aforementioned promising results cannot be generalized and may vary with different data-sets or queries, though their corresponding efforts have been rigorously evaluated. In fact, as the underpinnings of syntactic approaches to service discovery lie in the descriptiveness of service specifications, as a result WSDL documents without any proper comments or any representative keywords may deteriorate syntactic registries retrieval effectiveness [18–20]. A poorly written WSDL document, besides reducing its chances of being properly retrieved by a registry, hinders human discoverers' ability to understand and select the service afterward.

In spite of the intuitive implications of the use of poorly described WSDL documents against discovery, to our knowledge, there is a lack of studies that identify, measure and provide solutions to this problem. On the other hand, there is ongoing research on measuring the cost and benefits of bad and good API design practices [21]. However, as far as we know, software practitioners lack empirical evidence showing whether detected frequent API and component design practices occur in Web Services development or not, and whether any practices affect the discoverability of services. Moreover, the impact of these practices on human discoverers' ability to understand a WSDL document has been not experimentally evaluated.

This paper presents a study of frequent practices, from now on *bad practices*, found in a body of service descriptions written in WSDL version 1.1 that affect the discoverability of Web Services. Our research aims to provide solutions to those practices causing poorly described services in order to assist publishers in the creation of services that can be more effectively understood and discovered by human developers using syntactic registries. To achieve this goal, we exhaustively

1 UDDI, http://uddi.xml.org/.

2 WSDL, http://www.w3.org/TR/wsdl.

analyzed a body of 391 publicly available WSDL documents, and observed a list of *bad practices* that occur within this dataset. Subsequently, we have supplied each bad practice with a corresponding course of actions to correct it. Afterward, to facilitate the detection and solution of the observed bad practices, we have described each of them in a general way that includes a description of: the problem, its solution, and an illustrative example. Therefore, following the well-known definition of anti-pattern [22], in this paper we present a novel catalog of Web Service discoverability anti-patterns along with their occurrences within WSDL documents downloaded from the Web. This catalog not only subsumes previous works [9,10,23,24] on detecting common practices that hinder Web Services discoverability, but also presents solutions to them.

This contribution has been validated by performing two experiments. The first experiment consisted of comparing the retrieval performance of three syntactic approaches for Web Service discovery using the original body of WSDL documents versus using an improved one that resulted after manually correcting the found anti-patterns. Experimental results empirically show that when feeding the employed registries with the improved body of documents, they were more effective not only in retrieving more relevant services within a result list of 10 candidates, but also in ranking them first in the result list, versus the discovery performance resulting from using the original WSDL documents. Moreover, this experiment shows the individual impact of the anti-patterns.

For the second experiment, we conducted a survey among software engineering students and practitioners to analyze whether several improved WSDL documents were more intelligible than their original versions or not. The results provide more evidence that removing the identified anti-patterns is important for helping discoverers in having a better service understanding, which is crucial for discovery.

The main contributions of this paper are:

- a survey of frequent practices that hinder Web Service discoverability,
- a catalog of discoverability anti-patterns, which allows publishers to create more discoverable service descriptions or improve existing ones, and
- experiments showing that employing this catalog to remove anti-patterns from WSDL documents is beneficial to connecting publishers and discoverers.

The rest of this paper is organized as follows. The next section presents details about WSDL documents and describes the cornerstone of syntactic approaches to service discovery. Section 3 discusses related works. Then, Section 4 presents WSDL anti-patterns. Through Section 5 we survey the presence of anti-patterns in real Web Services. Later, Section 6 presents a detailed case study showing how to avoid several discoverability anti-patterns from a real WSDL document. Section 7 shows the implications of removing the anti-patterns. Lastly, Section 9 concludes the paper.

2. Background

WSDL is a language that allows developers to describe two main parts of a service: its functionality and how to invoke it. Following version 1.1 of the WSDL specification, conceptually the functional description reveals the service interface that is offered to the outer world. The latter part specifies technological aspects, such as transport protocol and network address (a.k.a. end-points). Discoverers use the functional descriptions to match third-party services against their needs and, in turn, they take under consideration the technological details for invoking the selected service. Therefore, in the rest of this section we will focus on explaining the part of a WSDL document that deals with describing the intended functionality of a service.

A WSDL document describes the service functionality as a set of *port-types*, which arrange different *operations* whose invocation is based on *message* exchange. Messages stand for the inputs or outputs of the operations, indistinctly. Moreover, WSDL also allows providers to describe one or more exceptions as ordinary messages called *faults*. Besides, the main elements of a WSDL document, such as port-types, operations and their messages, must be given unique names. Optionally, these WSDL elements may contain documentation as comments. Fig. 1 depicts a concrete service definition.

Messages consist of *parts* that transport data between consumers and providers of services, and vice-versa. Each message part is arranged according to specific data-type definitions. The XML Schema Definition (XSD)[3] language is employed to express the structure of the message parts. XSD offers constructors for defining simple types (e.g., integer and string), restrictions and both encapsulation and extension mechanisms to define more complex elements. For example, the WSDL document depicted in Fig. 1 contains the code needed for representing a complex data-type, called "CountryCodes" which is exchanged in the input message of "GetRate" operation. Alternatively, the XSD code might be put into a separate file and imported from the WSDL document or even other WSDL documents afterward.

Using representative names and properly commenting WSDL document elements is essential for discovering services through syntactic registries, apart from the well-known benefits of documenting software and using representative names; e.g., facilitating system maintenance [25]. Syntactic registries preprocess WSDL documents to extract terms from them. Commonly, the collection of terms gathered from a WSDL document consists of the names and comments of port-types, operations and in/out messages [9,18,19,26–28].

[3] XML Schema Part 0: Primer Second Edition, http://www.w3.org/TR/xmlschema-0/.

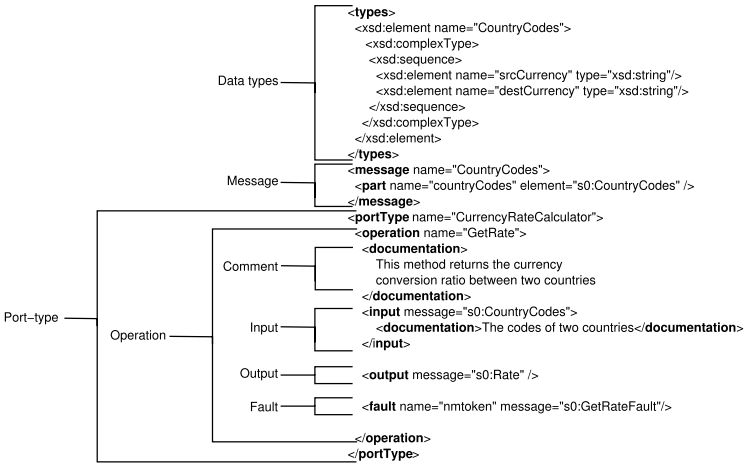


Fig. 1. Example of Web Service description using WSDL.

Once target terms have been extracted, syntactic registries usually preprocess them to split combined words (e.g., splitting "CountryCodes" leaves "Country" and "Codes", and "GetRate" causes "Get" and "Rate"), remove common non-relevant words, such as "the", "a", "for" or "by" (a.k.a. *stop-words*), bridge synonyms, and remove the commoner morphological and inflectional endings from words (a.k.a. *stemming*) [19]. Textual queries are frequently preprocessed in a similar way before matching them with the terms gathered from available service descriptions. Clearly, if a publisher builds unintelligible service descriptions that convey neither explanatory names nor explanatory comments, unveiling important information about the offered service functionality from the WSDL document will be a hard task, if possible.

To sum up, WSDL allows developers to describe the offered Web Service interface. Syntactic service registries commonly preprocess WSDL documents for extracting terms that may allow discoverers to find services relevant to their requests. With syntactic registries, looking for services related to a query heavily depends on how appropriate for discovery the WSDL documents and queries are. Therefore, the representativeness of names and comments within WSDL documents is crucial. Indeed how explanatory service descriptions are, influences service chances of being selected by the human discoverer who is using a discovery system. This is because this kind of service registry returns a list of candidate WSDL documents, which the user who performs the discovery must analyze. The intelligibility of WSDL documents plays a very important role here.

3. Related work

The problem associated with the quality of WSDL documents has been tackled from two perspectives. From a technological point of view, there are a handful of efforts to make Web Services more interoperable. These efforts examine current practices for developing Web Services, including the creation of WSDL documents. Using open standards does not mean all Web Services might be consumed by applications developed in any programming languages or platforms. Unconventional usage or errors in a WSDL document may degrade interoperability between providers and consumers. In [29] the authors examine different approaches for exchanging data between providers and consumers of services, and discuss the impact of these approaches on service interoperability. Similarly, the Web Services Interoperability Organization[4] is an open industry organization that defines profiles (agreements) that prescribe which options should be used in implementations of Web Service standards in order to guarantee interoperability.

Instead, there are four works that address this problem from the perspective of Web Service discovery. In [23] the author explains the impact of using "XSD wild-cards", when defining data-types, on the maintainability and discoverability of Web Services. A wild-card is a special XSD constructor; e.g, xsd:any and xsd:anyAttribute, which allows developers to leave one or more parts of an XML structure undefined. One detected reason for using such XSD constructors is to minimize the effort involved in modifying a service when it evolves, while assuring that consumers bound to old versions of the service will be able to correctly invoke and process the operations defined in its new version [23]. The idea is to include extension points in a WSDL document, by postponing the definition of some of its constituent parts. The author asserts that the main drawback of using such an extensibility mechanism is defining "vague interface contracts". A WSDL document is said to be vague when it defines at least one of its messages by means of XSD wild-cards, because there is no way to express which extensions are supported. Imagine an operation that defines its output using an xsd:any-based element, which means that the output can be any valid XML, then the output definition does not allow discoverers to exactly infer what the service response will be like. Therefore, [23] presents a different versioning strategy for WSDL documents, which removes XSD wild-cards from data-types, and facilitates Web Services forward compatibility at the same time.

4 Web Services Interoperability Organization, http://www.ws-i.org/default.aspx.

In [24] the authors present many difficulties that six students of a SOC course encountered while developing a large Web Services-based application. Some of the identified difficulties relate to understanding third-party Web Services. Specifically, the authors show that unclear "control parameters" within data structures and long identifier names make a Web Service harder to be understood [24]. A message associated with a control parameter, besides carrying data objects, includes miscellaneous objects, such as a log object. As this kind of objects tends to control the execution flow of service consumers, the authors refer to them as "control parameters". Control parameters are frequently used to inform about errors that occur during the execution of an operation. Moreover, the authors observe that all six users either misread names or were confused by them, and all complained about the length of the names.

In [10] the authors propose to crawl public registries on the Internet to gather existing Web Services, in order to analyze their documentation. To this end, the authors wrote several crawlers that fetch: the names, providers, registration information (a textual comment describing the service being published) and WSDL documents of Web Services that were published in five registries on the Web. Initially, the authors gathered 2432 Web Services, but this number decreased to 1544 because some registered services did not have a valid URL to their associated WSDL documents. Moreover, the authors discarded those Web Services that were duplicated or had no valid WSDL file entry; i.e., the WSDL document was not well-formed or it did not conform to the WSDL standards. Thus, the number of gathered services fell to 640. Then, the authors analyzed the lengths of the textual comments of the 640 services (including the registration information and all the comments present in the WSDL files). The paper shows that 80% of the textual comments have less than 50 words and 52% of the WSDL documents have less than 20 words. Afterwards, the authors averaged the lengths of the textual comments associated with <operation> elements in the WSDL documents. In the employed collection, the average comment length per operation of nearly 80% of the services has less than 10 words, and almost 50% of the services have no documentation for any of the offered operations.

One difference between our work and the three previously described is that though the latter have meticulously described the poor practices found, their implications on the retrieval process of service registries have not been corroborated experimentally. Instead, in [9] the authors detect "naming tendencies" in WSDL documents and empirically show that these tendencies negatively impact the retrieval effectiveness of a syntactic registry. The authors analyze the names of message parts that belong to 596 WSDL documents, which were gathered from Internet repositories. They divide in/out messages into crumbled parts. Afterwards, the name of each part is compared with the rest of part names. As a result, the authors have identified four naming tendencies that take place in the observed service parts. Broadly, these tendencies show that developers use common phrases within part names. For example, the authors have found that a message part standing for a user's name, is called "name", "lname", "user_name" or "first_name" [9]. Moreover, they have found that abbreviations and names shorter than three characters were ineffective in matching part names. Finally, the paper empirically shows that the retrieval effectiveness of a syntactic registry can be improved by enhancing its underlying matching approach for dealing with the observed tendencies.

This paper explicitly addresses the quality of WSDL documents from the perspective of a discoverer, pursuing recurrent problems that attempt to prevent the understandability and discoverability of a service. This paper presents a catalog of eight bad practices that frequently occur in a body of public WSDL documents. This catalog not only subsumes the problems related to "XSD wild-cards", "control parameters", "poor documentation" and "naming tendencies", which were previously identified in [9], [10], [23], and [24] respectively, but also supplies each problem with a practical solution. In this sense, this paper describes a case study that shows how to improve a WSDL document. Moreover, this paper presents novel experiments that provide hints on how the identified problems impact on the effectiveness of different approaches to service discovery and on human discoverers' ability to understand third-party services as well.

4. Web Service discoverability anti-patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. Anti-patterns extend the notion of patterns to express obvious, and inappropriate solutions to recurrent problems that have been supplied with refactored solutions that are clearly documented, proven and repeatable. Building a catalog of design anti-patterns means representing design errors, which are known to occur frequently in a large number of applications, in order to enable their detection and solution.

Through this section, each identified bad practice is studied to provide a sound and practical solution. Table 1 presents a comprehensive list of the resulting anti-patterns using the following template:

Name A succinct name to convey the essence of the anti-pattern.

Concern A classification of the anti-pattern related to: problems on how a service interface has been designed, problems on the comments and identifiers used to describe a service, and problems on how the data exchanged by a service are modeled. We refer to these three types of concerns as *Design, Documentation* and *Representation*, respectively. In the context of WSDL, the Design concern deals with the organization of port-types and operations. The Documentation concern deals with comments and names associated with port-types, operations and messages. Finally, the Representation concern deals with type definitions.

Problem The commonly occurring bad practice that relates to the anti-pattern.

Table 1
Web Service discoverability anti-patterns.

Anti-pattern	Concern	Problem	Manifest	Suggested solution
Inappropriate or lacking comments	Documentation	Occurs when: (1) a WSDL document has no comments, or (2) comments are inappropriate and not explanatory.	(1) Evident, or (2) Not immediately apparent	Create explanatory comments and place them in the correct part of the WSDL document.
Ambiguous names [9]	Documentation	Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent	Replace ambiguous or meaningless names with representatives names.
Redundant port-types	Design	Occurs when different port-types offer the same set of operations.	Evident	Summarize redundant port-types into a new port-type.
Low cohesive operations in the same port-type	Design	Occurs when port-types have weak semantic cohesion.	Not immediately apparent	Divide non-cohesive operations into different port-types.
Enclosed data model	Representation	Occurs when the data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD ones.	Evident	Move data-type definitions from WSDL documents to XSD files.
Redundant data models	Representation	Occurs when many data-types for representing the same objects of the problem domain coexist in a WSDL document.	Evident	Summarize redundant data-types into a new data-type.
Whatever types [23]	Representation	Occurs when a special data-type is used for representing any object of the problem domain.	Evident	Replace Whatever types with data-types that properly represent the required objects.
Undercover fault information within standard messages [24]	Design	Occurs when output messages are used to notify service errors.	Present in service implementation	Use fault messages for conveying error information.

Manifests Another classification based on how an anti-pattern manifests itself. An anti-pattern is *Evident* if it can be detected only by analyzing the structure, or syntax of the WSDL document. *Not immediately apparent* means that detecting the anti-pattern requires not only a syntactical analysis but also a semantic one. Finally, *Present in service implementation* anti-patterns may not show themselves in the WSDL document, thus requiring the execution of the associated service to be detected.

Suggested solution The refactored solution that solves the problem.

4.1. Inappropriate or lacking comments

Commonly, WSDL documents are not well documented [10], or worse some WSDL documents are not documented at all. Placing explanatory comments within a WSDL file makes the intended functionality of its associated service easier to understand. Furthermore, syntactic service registries exploit this kind of documentation, present in WSDL documents, to support discovery.

When a WSDL document has no comments, we say that it evidently suffers from Inappropriate or lacking comments anti-pattern. However, inappropriate comments may be not immediately apparent. A WSDL document is said to be well documented when each of its operations has a concise and explanatory comment, which describes the semantics of the offered functionality [30]. Moreover, as WSDL allows providers to comment each part of a service description separately, then a good practice is to place every <documentation> tag in the most restrictive ambit possible. For instance, if the comment refers to a specific message, it should be placed in that message and not in the operation that uses the message. Instead, a badly commented operation typically includes information that is not directly related to its functionality; e.g., details about either the authors or licenses of the service.

The solution to the Inappropriate or lacking comments anti-pattern is to comment the different parts of a WSDL document. The left side of Fig. 2 depicts a non-commented WSDL document for a service that translates a given text from English to German, whereas the right side shows the same WSDL document after being improved. In Fig. 2, each documentation element is underlined. As a result of the refactoring, the WSDL document not only is more intelligible, but also conveys more relevant terms, which is essential for syntactic registries such as [18,19], and many measures for assessing the similarity between Web Services [26]. With this example, specifically, by preprocessing terms that were extracted from the documentation elements of the improved WSDL document, we gathered 4 occurrences of ‘‘translat’’ stem, 3 of ‘‘text’’ and 2 of both ‘‘english’’ and ‘‘german’’, respectively. This means that these comments allow extracting more terms related to the functionality of the service.

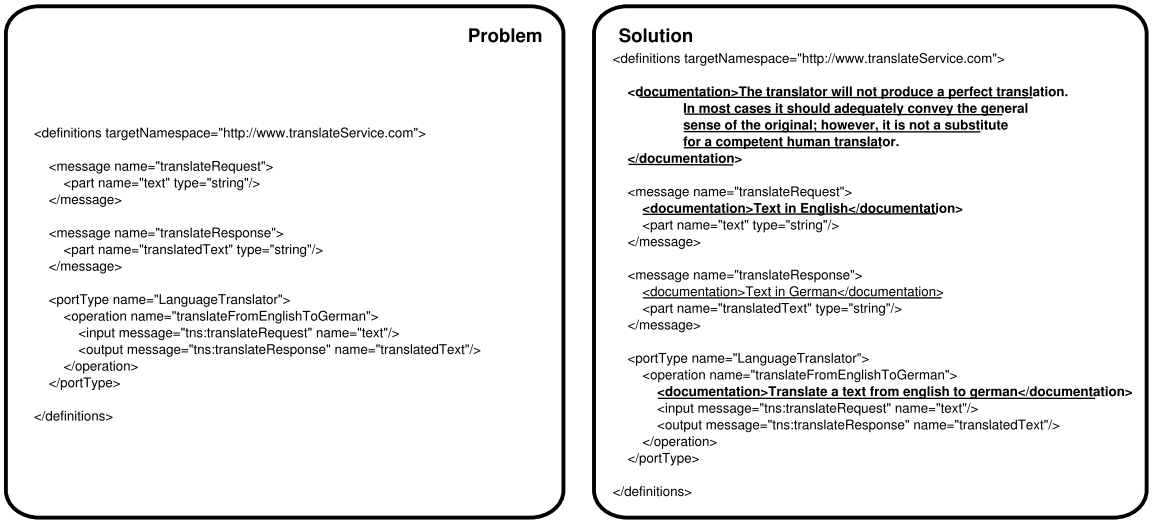


Fig. 2. Example of Inappropriate or lacking comments anti-pattern.

4.2. Ambiguous names

Another common bad practice is to use meaningless or cryptic terms to name port-types, operations, messages, and part elements. The name of a WSDL element is used not only as a unique identifier, but also as a descriptor. From a discoverer's point of view, a representative name should describe the semantics of an element, thus syntactic approaches to service discovery gather and preprocess them.

Representative names should conform to some semantic and syntactic characteristics. Semantically, a representative name should describe what its element represents, then meaningless names, such as in0, arg1 or foo, should be avoided. Moreover, if there are two or more elements within a WSDL document standing for the same concept, these elements should be equally named. For instance, if an operation receives user's details as input and another operation produces user's details as output, their corresponding message parts should have the same name.

Syntactically, on the other hand, the name of an operation should be in the form: <verb> "+" <noun>, because an operation is an action [31]. In the case of a message, a message part, or a data-type, its names should be a noun or a noun phrase; otherwise it may mean that a message conveys control information. This case is related to the Undercover fault information within standard messages anti-pattern, which will be described in Section 4.8.

Moreover, as syntactic registries rely on popular naming conventions, such as JavaBeans or Hungarian notations, to split long names [18,19,27,26], if a name is composed by two or more words, the name should be written according to common notations. For example, the name "thisisthenameofanelement" should be rewritten as "thisIsTheNameOfAnElement" or "this_is_the_name_of_an_element". Besides, the latter names are clearly easier to read than the original ones. Another consideration is the length of a name. A name should be neither too short nor too long. A recommended length is between 3 and 15 characters [9].

The solution to the Ambiguous names anti-pattern is to replace meaningless names with names that follow the conventions mentioned before. Moreover, names in the refactored WSDL document should be consistent. It means that the names in the WSDL document and the concepts represented by these names should have an univocal correspondence. As a result, a refactored WSDL document is free from ambiguous names. Moreover, a refactored WSDL document allows syntactic registries to extract more terms related to its intended functionality and its problem domain.

4.3. Redundant port-types

Another common bad practice is to repeat port-types. As we will show in Section 5, we have found that there are redundant port-types in 60% of the documents within the analyzed data-set. A redundant port-type is one that consists of the same set of operations that are offered by another port-type of the same Web Service. Although repeating the interface of a service might seem like a weird thing to do, developers typically define two or more port-types that offer the same operations with the same messages, but each port-type is bound to a particular transport protocol. Moreover, typically the names of redundant port-types start by describing the offered functionality, but each of them ends with a different abbreviation that represents a concrete transport technology. In the end, this springs unnecessarily big and puzzling WSDL documents.

For example, the left side of Fig. 3 depicts a service for translating a given English text into German. This translation service is defined as having two "different" port-types for consuming the service with either SOAP or HTTP, respectively.

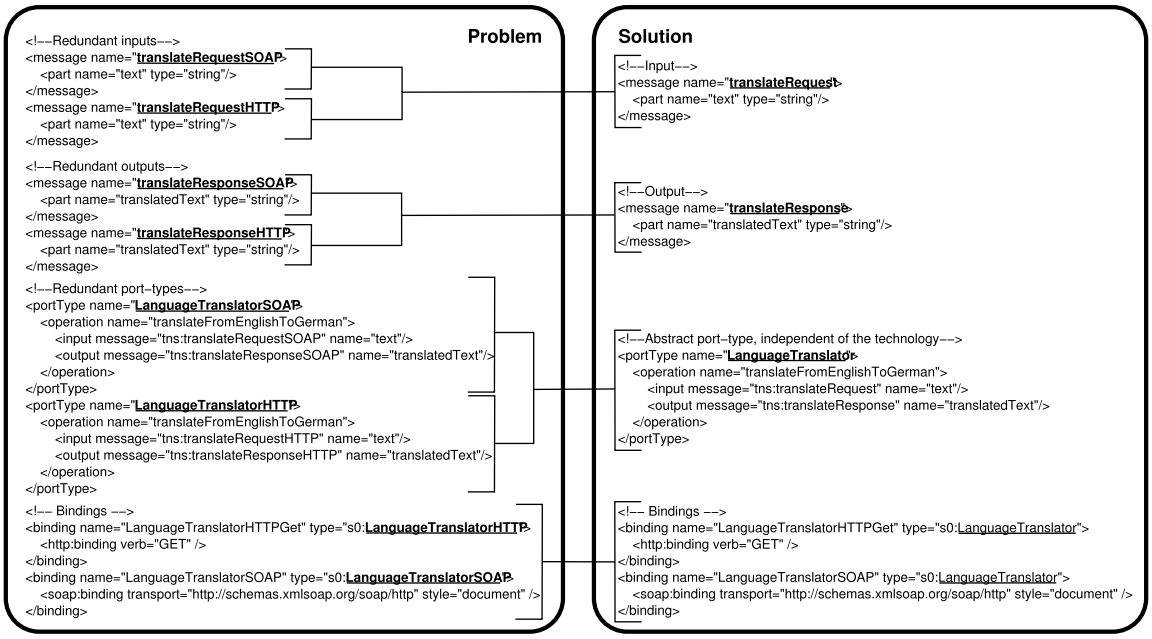


Fig. 3. Example of Redundant port-types anti-pattern.

In this example, "LanguageTranslatorSOAP" and "LanguageTranslatorHTTP" port-types define the same set of operations. Indeed, they define the same messages.

Redundant port-types may be considered as an attempt to influence the ranking of Web Services returned by a syntactic registry, apart from the obvious obstacle that this represents to isolate the potential consumers of a component from the implementation details [32]. Commonly, syntactic registries determine that a term is important for a WSDL document if it often occurs in that document; i.e., the higher the frequency of the term within a document is, the higher its importance becomes [18,19,26]. Based on the importance of gathered terms, syntactic registries rank Web Services similar to a given query. When publishing WSDL documents in a syntactic registry, if providers repeat the definition of the offered interfaces, the registry will gather more occurrences of relevant terms. As a result, Web Services with redundant port-types will have more chances of being ranked first. This situation is analogous to what has been described as Google Bombing.[5]

The solution to the Redundant port-types anti-pattern is to combine redundant port-types into a single one that does not include technological aspects. The refactored code for the translation example is shown in the right side of Fig. 3, in which solid lines emphasize the relations between the elements of the original WSDL document and their refactored counterparts. Basically, the refactoring consists in removing redundant port-types and messages, keeping one of each kind, naming them with protocol-independent names and updating the definitions of the concrete bindings. Accordingly, if the refactored WSDL document is free from redundant code, then a syntactic registry will not extract redundant terms from it. Besides, the improved WSDL document will be more concise than the original one.

4.4. Low cohesive operations in the same port-type

Another common bad practice is to arrange non-cohesive operations in a single port-type. In the context of Web Services, cohesion is a measure of how strongly-related the operations within a port-type are. If the operations grouped by the port-types of a Web Service tend to belong to the same domain or jointly provide a set of semantically related functions, the service is said to have high cohesion. Instead, a port-type with weak cohesion, or non-cohesive, has operations for performing semantically unrelated functions and even from different problem domains. In structured design, modules with high cohesion tend to be preferable [32].

Weak cohesion often occurs in Web Services that place operations for checking the availability of the service and operations related to its main functionality into a single port-type. An example of this bad practice is to include operations such as "isAlive", "getVersion" and "ping" in a port-type, though the port-type has been designed for providing operations of a particular problem domain; e.g., to give exact information on commodities. From the perspective of syntactic approaches to service discovery, WSDL documents with highly-cohesive port-types convey terms that are representative of the domain of their container services mostly. Then, when asking a syntactic registry using queries comprising terms related to the domain of the services, these services will have higher probabilities of being discovered.

[5] Google Bombing, http://en.wikipedia.org/wiki/Google_bomb.

The solution to the Low cohesive operations in the same port-type anti-pattern is to remove the non-cohesive operations from their common port-type and put them into either a new port-type or a new Web Service. The idea is to keep only related operations within the original port-type, to increase its cohesion. At this point, if non-cohesive operations are defined in a new port-type, the Low cohesive operations in the same port-type anti-pattern will occur in the new port-type. Therefore it may be required to iterate the proposed solution until neither the original port-types nor the new port-types contain non-cohesive operations. In the end, a refactored WSDL document contains more port-types, but they are more cohesive than the port-types in the original WSDL document.

4.5. Enclosed data model

Another detected bad practice is to confine ad hoc data model definitions in each service description that requires them. An enclosed data model is a data model that is placed within a WSDL document, and can be used *only* from the operations described in their container WSDL document. Instead, an imported data model is developed in isolation from a service description, placed within a separate XSD document, and referenced from WSDL documents as required. With respect to Web Service discoverability, we assume that developers use best practices for naming and modeling data-types, when building their data models within separate XSD files. This assumption is usually true, because imported data models are written to be reused by other developers. As syntactic registries extract relevant terms from data-types associated with messages [19,27,28], data models conceived for being reused may positively impact the precision of this kind of registries.

Imagine a Web Service for checking stocks when the stock market is open and another service for checking stocks when the stock market is closed. The two services offer an operation that retrieves market information, the only difference is the moment when that information is collected. Suppose that the data-types of one service are modeled following an enclosed approach, thus the corresponding WSDL document contains a data-type definition named "StockQuote" that stands for market information. In the same way, the developers of the other service define a similar data-type, but named "StockInfo", within their WSDL document. Although the data-types "StockQuote" and "StockInfo" represent the same concept, their definition code is not reused. However, if we keep only one of these data-types, improve the names of its attributes and comments, place it in a separate schema document, and in turn, reference it from both WSDL documents, then the data model is reused.

The solution to the Enclosed data model anti-pattern is to use best practices for naming and modeling data-types, defining them in separate XSD documents and, in turn, combining separate model definitions as required using the tags "include" or "import". This allows publishers to share and reuse data models rather than re-design ad hoc data-types for each new Web Service. Additionally, reusing data-models ensures that some ambiguous names (see Section 4.2) will be avoided, which states that element names should be consistent. It is worth noting that when data-types are not going to be reused or are very simple, they can be part of the WSDL document to make it "self-contained", but they should be designed following best practices for naming and modeling data-types.

4.6. Redundant data models

Defining the same data-type two or more times is another common bad practice. Suppose a developer combines the "enclosed versions" of the Web Services for checking stocks at days' open and at days' close into the same WSDL document. Then, two data-types for representing the same object of the problem domain, "StockQuote" and "StockInfo" will coexist in the WSDL file. In general, a redundant data model occurs when, at least, two data-type definitions stand for the same exchangeable information in a WSDL document. For a syntactic registry, redundant data models may produce the same effect as redundant port-types. Besides, this anti-pattern, like the Redundant port-types anti-pattern, causes unnecessarily big and puzzling WSDL documents from the perspective of a human discoverer.

The Redundant data models anti-pattern is a problem often caused by a bad representation of the required data-types. The refactoring strategy to correct this anti-pattern consists in replacing redundant data-types with a single data-type, and changing references to the old data-types in message parts for references to the refactored one. In consequence, the refactored version of the service description is free from redundant XSD code. Then, the service definition may be conciser and easier to be understood by third-party discoverers than its original version. Optionally, the refactored data model should be defined in a separate XSD document to prevent occurrences of the Enclosed data model anti-pattern.

4.7. Whatever types

Another common practice found in WSDL documents is to use general purpose data-types for representing any object of a problem domain. We refer to this kind of data-types as Whatever types. Typically, a Whatever type relies on the use of the XSD constructs xsd:any and xsd:anyAttribute. Below we present the XSD code for defining a data-type, called DataSet, which developers frequently use for exchanging a sequence of almost any complex XML structure between providers and consumers:

```
<xsd:element name="DataSet" nillable="true">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="s:schema"/>
         <xsd:any/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

A DataSet instance can be any valid XML, even an empty XML. While this might be seen as an alternative for building extensible operations [23], it obscures the domain and range definitions of the operations that convey this kind of types in their messages. In consequence, this data model design anomaly makes operations hard to be understood by third-party discoverers. As an example, suppose a port-type named "RandomGenerator" comprising one operation whose signature is "generate():DataSet". The signature and data-type of this operation are ambiguous. A human discoverer cannot determine what the structure of the operation response will be like until they invokes the operation, and neither whether he/she will receive more than one object. In general, when using Whatever types there is no way to express which extensions are supported [23]. Moreover, this kind of types typically does not carry explanatory terms, thus hindering discoverability through syntactic registries.

Instead, suppose another operation, whose signature is "generate():RandomNumber" with RandomNumber being:

```
<xsd:simpleType name="RandomNumber">
  <xsd:restriction base="xsd:double">
    <xsd:minInclusive value="0"/>
    <xsd:maxExclusive value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

From the signature of this new operation it may be derived that the operation returns one random number. Moreover, as RandomNumber extends the double primitive data-type and limits its range between 0 and 1, a human discoverer may derive that the operation returns a random number represented as a double between 0 and 1.

The Whatever types anti-pattern can be symptomatic from bad data model designs or the desire to make Web Service interfaces extensible [23]. When the first problem occurs, the solution associated with this anti-pattern comprises replacing all "Whatever" definitions with proper data-types and updating references to them. A proper data-type should be intended to merely convey either the information that an operation needs as input or produces as output, and should be named with a representative name. In the end, the refactored operation and its message data-types will convey relevant terms rather than "DataSet". Instead, if this anti-pattern stemmed from extensible interfaces, the solution is to apply the versioning strategy proposed in [23]. Broadly, this versioning strategy recommends to build new backward compatible data-types by using the extension mechanism of XSD, which allows developers to define a new data-type from one created previously.

4.8. Undercover fault information within standard messages

Using output messages to inform about errors that occur during the execution of an operation is a common bad practice. In consequence, an output message is designed for exchanging error information when the operation fails, or a result value when the operation successfully finishes its execution. Clearly, to do this the output message of an operation must be defined using a flexible data-type. This, besides causing occurrences of the Whatever types anti-pattern, obscures the underlying logic of an operation. Returning fault information within output messages may negatively impact syntactic registries that exploit either the names or the XML structure of message parts, such as [18,19,27] and [33], respectively.

This anti-pattern normally appears when a message is associated with a general purpose data-type, e.g., the DataSet type discussed previously. Another frequent form of this anti-pattern requires a message definition of three parts: a part informs whether an error occurred, another part conveys error details if any, and the third part carries the operation result. For example, following this "three-parts" approach the output message of an operation for calculating the factorial of a non-negative integer given as input is:

```
<message name="calculateFactorialResponse">
    <part name="isError" type="xsd:bool"/>
    <part name="stackTrace" type="xsd:string"/>
    <part name="factorial" type="xsd:long"/>
</message>
```

The part "isError" communicates whether an error occurred. If an error occurs, then the part "stackTrace" will convey its description, otherwise it will convey no data. On the other hand, if there is an error, "factorial" will be empty, if not it will transport the calculation result. Furthermore, now the output message not only transports application data, but also tells the client what to do. This is an special case of control coupling, a situation that should be avoided in structured design [32]. From the perspective of syntactic approaches to service discovery, the presented control parameter conveys the terms: "error stack trace factorial", in which 75% of them are unrelated to the factorial calculations domain.

The solution to the Undercover fault information within standard messages anti-pattern relies on WSDL support for handling error information. Then, to solve this anti-pattern, error information should be exchanged within fault messages. Returning to the factorial calculator example, the refactored code is:

```
<message name="calculateFactorialFault">
    <part name="errorDescription" type="xsd:string"/>
</message>
<portType name="FactorialCalculator">
    <operation name="calculateFactorial">
        <documentation>Returns the factorial calculator
              for a given number</documentation>
        <input message="tns:calculateFactorialRequest" name="number"/>
        <output message="tns:calculateFactorialResponse" name="result"/>
        <fault message="tns:calculateFactorialFault" name="errorDetails"/>
    </operation>
</portType>
```

The refactored operation has three messages: input, output and fault messages. Now, input and output messages only exchange application data between client and server. Additionally, the fault message has been specially designed for supplying invokers with a description of errors. In this example, as error information is a textual description, the fault is associated with a string. In other contexts, it could be associated with a complex-type for representing the stack-trace of the service operation. Finally, the refactored WSDL document is free from general purpose types and control coupling.

5. Data-set analysis

This section discusses the frequency of the discoverability anti-patterns in public WSDL documents. We have analyzed a body of WSDL documents that were gathered from Internet repositories by Hess et al. [34]. This data-set consists of 391 WSDL documents. Initially, we manually revised 130 files of the data-set and documented a detailed list of bad practices that frequently occur within this subset. Accordingly, we have developed the catalog of WSDL discoverability anti-patterns of Section 4. Then, to have an assessment of how common these bad practices are, we manually analyzed the whole body of WSDL files. Specifically, on average, an anti-pattern is present in 40% of the WSDL documents, and 82% of these documents are affected by, at least, one anti-pattern.

For analyzing the WSDL documents of the data-set we developed a detection criterion for each WSDL anti-pattern. As described in Section 4, detecting an anti-pattern is strongly related to the way it manifests itself. Thus, we have identified that some anti-patterns are harder to detect than others. Specifically, detection criteria for Evident anti-patterns are based on the syntax of a WSDL document. A clear case of this is Enclosed data model anti-pattern, since it can be deterministically detected by applying the following rule: "if at least one type is defined in a WSDL document, then the anti-pattern occurs". Another clear case of this, is when a WSDL document has no comments.

Detection criteria for Not immediately apparent anti-patterns require to analyze not only the syntax of a document, but also its semantics comprising questions like "Is the name of this message part ambiguous?" or "Is the documentation of this operation clear enough?". The answers to this kind of questions depends on personal judgement. However, analyzing whether the names and comments present in a WSDL document follow the conventions described in Section 4.2 may help to detect Ambiguous names and Inappropriate comments. In addition, sometimes Undercover fault information within standard messages anti-pattern has no footprint in a WSDL document. If a message includes a part to inform whether an error has occurred, this anti-pattern can be detected. Instead, if an output message part is called "parameter" and it exchanges a Whatever type, it might be used to inform an error, but it is impossible to know for certain. In this case, the Undercover fault information within standard messages anti-pattern cannot be detected, unless the service implementation fires an error during a request. For the study presented in this section we considered the Undercover fault information within standard messages anti-pattern only if it can be detected in the WSDL document.

The results showed that some anti-patterns affect more WSDL documents than others, but even the least frequent anti-pattern occurs in 31 WSDL documents. Graphically, each bar of Fig. 4 depicts the number of WSDL documents that suffer from an anti-pattern. It is worth noting that these values represent the number of WSDL documents affected by an anti-pattern, not the number of anti-pattern occurrences. For example, if a WSDL document has 2 occurrences of the Redundant data models anti-pattern, we count as only 1 affected file. The reason to present the results in this way, is that some anti-patterns occur more than once in a WSDL document (Inappropriate or lacking comments, Ambiguous names, Redundant data models, Whatever types, Undercover fault information within standard messages), but other anti-patterns occur only once in a WSDL file (Redundant port-types, Low cohesive operations in the same port-type, Enclosed data model). Therefore, assessing the number of service descriptions that suffer from each anti-pattern depicts how important each anti-pattern is for the surveyed data-set.

Notably, though good naming and commenting practices facilitate software reuse and the fact that Web Services are developed for reuse, the fractions of documents that suffer from Ambiguous names and Inappropriate or lacking comments anti-patterns are 82% and 69%, respectively. Additionally, although enclosed data-type definitions hinder the reuse of data models, the Enclosed data model anti-pattern occurs in 70% of the documents. Similarly, notwithstanding port-types are meant to define the functionality of a service independently of any technological aspect, we have found that there are redundant protocol-dependent port-types in 60% of the documents in this data-set. On the other hand, the least common anti-pattern within this data-set is the Low cohesive operations in the same port-type anti-pattern, which affects 7.6% of the documents. Likewise, the proportion of Web Service descriptions that suffers from Undercover fault information within standard messages anti-pattern is 10%.

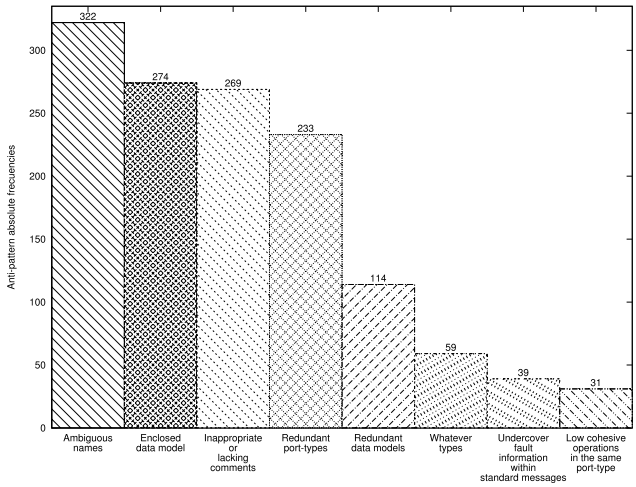


Fig. 4. Anti-pattern occurrences within 391 Web Services.

In addition, we have observed that the occurrence of some anti-patterns may be related. For instance, Whatever types or Redundant data models anti-patterns never occur without Enclosed data model anti-pattern. Probably, this happens because if the developers of the service take time to separate a data model from WSDL documents, they will want to reuse that data model in other services. Therefore, in terms of understandability and discoverability, developers will strive to produce better models. However, it may be possible that the anti-pattern has not been detected because separating the data model from the WSDL helps conceal others anti-patterns. This could be the case of some types defined in two different XSD files, i.e., a redundant data model, and then imported from a WSDL document.

Another case where anti-patterns are related is the Uncovered fault information within standard messages anti-pattern with the Ambiguous names anti-pattern. As shown in the example in Section 4.8, to inform that an error has occurred in an output message may require control coupling [32]. The names of the parameters, when this kind of coupling is present, usually take the form of <verb>+<noun>, e.g., "isError". And that is an instance of the Ambiguous names anti-pattern.

6. A case study: A real WSDL before and after improving its discoverability

This case study shows how some of the anti-patterns manifest in a real life Web Service description, how to apply their proposed remedies and how the resulting WSDL document is. For the case study, we selected a WSDL document[6] from the data-set described in Section 5. Then, we used the criteria and the remedies described in Table 1 to identify and then correct the anti-patterns that take place in this WSDL file. Finally, we analyzed, from a discoverer's point of view, the implications of improving the WSDL document following the catalog recommendations.

The functionality of the selected Web Service is to convert a force measure given in some unit, such as dyne, gram-force or newtons, to another unit. This service defines one operation, named "ChangeForceUnit" that receives as input a force value, its force unit together with a target force unit, and returns a force value as output. The part of the WSDL document that describes this operation is shown in Fig. 5. The figure has two boxes, the left one shows the messages and port-types of the WSDL document, while the right one shows the enclosed XSD code. An outstanding characteristic of this WSDL document is that although it defines only one operation, it has three port-types and six messages.

Fig. 5 shows the WSDL document along with the anti-patterns that occur in it. First, there are four Evident anti-patterns in this WSDL document: Enclosed data-model, Inappropriate or lacking comments, Redundant data-models and Redundant port-types. The Enclosed data model anti-pattern is present because the data-type definitions are in the WSDL document file. Because the WSDL document has no comment, it is evident that the Inappropriate or lacking comments anti-pattern is present. The Redundant data models anti-pattern, shown in the XML Schema box in Fig. 5, is a consequence of defining a data-type for "double", which is already defined as an XSD primitive type. Finally, the Redundant port-types anti-pattern occurs because all port-types in this WSDL document have only one operation, named "ChangeForceUnit", with three parameters as input and one as output. Furthermore, all port-type names follow the pattern "ForceUnit" plus transport technology name, e.g., ForceUnitSoap, ForceUnitHttpGet and ForceUnitHttpPost. This means that all port-types stand for the same interface, but they are redefined because of their underlying transport protocol.

On the other hand, one Not immediately apparent anti-pattern affects this WSDL document. This anti-pattern is Ambiguous names anti-pattern; however, to confirm this it is necessary to analyze the name of each WSDL element. For

6 WSDL document for the case study, http://www.webservicex.net/ConvertForec.asmx?WSDL.

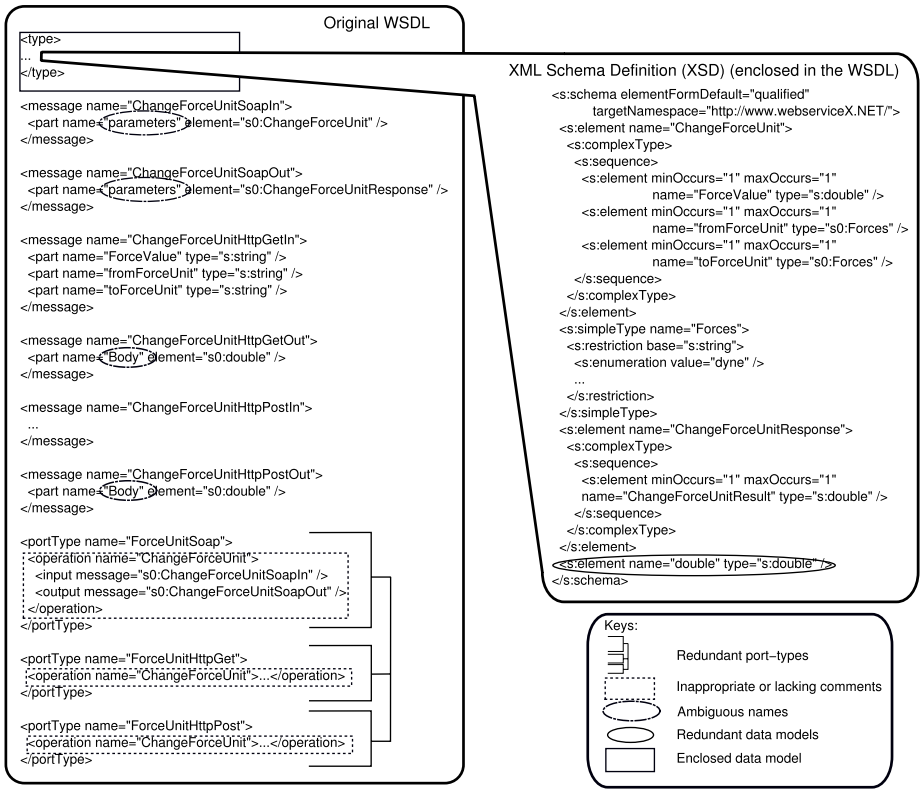


Fig. 5. Original WSDL document.

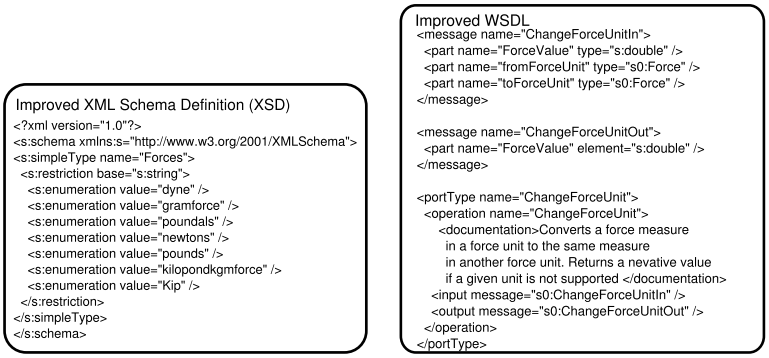
instance, names like ‘‘parameter’’ or ‘‘body’’, present in this WSDL file, are too general. The names should be representative of the service functionality, i.e., performing force unit conversions.

In order to create an equivalent WSDL document without anti-patterns is enough to apply all the anti-pattern remedies in any order. However, it is recommendable to apply the remedies of Enclosed data model first, Redundant port-types and Redundant data models anti-patterns second and finally the other anti-pattern remedies. The reason to fix the Enclosed data model first is that its associated remedy causes a smaller WSDL file. Then, as the remedies associated with Redundant port-types and Redundant data models anti-patterns eliminate redundant elements, the anti-patterns that are conveyed in these elements will be eliminated as well, i.e., the Ambiguous names anti-pattern.

Returning to the case study, the improvement process starts by removing the Enclosed data model anti-pattern by separating the XSD code from the WSDL document and then importing the former into the WSDL document. Then, the Redundant data models anti-pattern can be remedied by deleting the redefinition of ‘‘double’’ and replacing all references to ‘‘s0:double’’ with references to ‘‘s:double’’, which is the primitive type. In order to remove the Redundant port-types anti-pattern it is necessary to identify all repeated instances of the port-type and remove all of them, leaving only one instance in the WSDL document. The next step is to remove all messages that are not referenced by any operation. Then, the messages and the port-type must be renamed removing all reference to transport protocols and, finally, all binding elements must be changed to reference the new port-type. Since this new port-type is transport protocol independent, the Redundant port-types anti-pattern problems are not present in the improved WSDL document.

Having remedied all the discoverability anti-patterns that occurred in the analyzed WSDL document, we will inspect the resulting document. Fig. 6 shows the improved WSDL document. The first characteristic of the improved WSDL file is that it imports the XML Schema because the XSD code has been removed from the WSDL file. Another characteristic is that the new WSDL document is shorter than the original one. However, the number of relevant words, such as ‘‘unit’’, ‘‘force’’, ‘‘measure’’ or ‘‘change’’ in the improved WSDL document is higher than in the original. Moreover, this WSDL document has no too general words, e.g., ‘‘parameter’’ or ‘‘body’’. Note that a refactored WSDL document is not always shorter than its original version. However, applying the anti-pattern remedies always increases the number of relevant words and reduces the number of non-relevant ones. As shown in Fig. 6, the improved WSDL document only contains the information needed to communicate the functionality offered by its service.

The absence of any comment in the original WSDL document may hinder the discovery and correct use of this service. For instance, a syntactic registry has less words to compare the service with a query. Furthermore, a human discoverer is unable to know whether the service uses a negative value output as an error signal. For these reasons, all the operations of the improved WSDL document are properly commented. Another problem with the original WSDL document is that some



(a) XML schema definition. (b) WSDL document.

Fig. 6. Improved WSDL document and XML Schema.

element names are ambiguous. In the input message of the offered operation, the name of a part is ''parameters''. Therefore, we obtained more explanatory names by separating this part in three. Besides, the naming problem is also present in the output message part. In this case, an appropriate name for this part is ''ForceValue'' because it represents the same concept as the input message part with the same name.

The case study shows that applying the anti-pattern remedies to a WSDL document adds relevant words to it and removes irrelevant ones. Moreover, the case study also illustrates that all remedies can be applied in any order. However, the order might affect the effort required to apply them. For instance, if a WSDL document suffers from both Inappropriate or lacking comments anti-pattern and Redundant port-types anti-pattern, applying the Redundant port-types anti-pattern remedy first would reduce the number of non-commented or inappropriately commented operations.

7. Experimental results

As described through Section 4, the identified anti-patterns may hinder service chances of being discovered through approaches to service discovery that rely on meta-data present in WSDL documents. Furthermore, these anti-patterns may hinder a discoverer's ability to understand discovered services. We performed different experiments to provide empirical evidence of the impact of each individual anti-pattern on the retrieval effectiveness of several discovery systems. Additionally, we conducted a survey about how software developers perceive two versions of the same services: one version with anti-patterns and another without anti-patterns.

Broadly, the first experiment consisted in comparing the effectiveness of three approaches to service discovery when using different versions of a data-set of WSDL documents. From now on we will refer to the employed discovery approaches as service registries, indistinctly. The applied versions of the body of WSDL documents were built by removing anti-patterns. One version consisted of the original WSDL documents. Another version consisted of the WSDL documents without anti-patterns, i.e. after removing all found anti-pattern occurrences. Moreover, we built one version of the data-set per anti-pattern, in which the corresponding anti-pattern was removed from all the WSDL documents. The results empirically confirmed that using the data-set version without anti-patterns, the employed registries performed better in retrieving relevant services. Furthermore, these results empirically pointed out that removing the Inappropriate or lacking comments anti-pattern contributed to the effectiveness of the employed service registries more than removing the other anti-patterns.

The second experiment consisted in asking software developers and software engineering students, who were taking a SOC[7] course at the UNICEN, about the ability of several WSDL documents to explain what the functionality offered by their corresponding services was. We compared the answers related to WSDL documents with anti-patterns versus the answers related to improved WSDL documents. The results shown a major tendency in the answers, specifically 84.62% affirmed that the improved WSDL documents were more intelligible than the original ones. The next two subsections describe the experiments related to discovery and intelligibility, respectively.

7.1. Measuring the impact on discovery through syntax-based registries

The methodology followed to perform the experiment comprised publishing each version of the data-set in the service registries, performing queries and analyzing whether the retrieved services were relevant to the queries or not. Although the employed service registries are different, they return an ordered list of WSDL documents relevant to a given query, being the WSDL document at the top of the list the most relevant, and so on. Therefore, several metrics were used to evaluate the impact of removing identified anti-patterns, in terms of the proportion of relevant services in each retrieved list and their positions relative to non-relevant ones.

[7] SOC course at the UNICEN, http://www.exa.unicen.edu.ar/~cmateos/cos.

The registries used to perform the test were built from Lucene [35], Web Service Query By Example (WSQBE) [19] and IR Lexical Structural (ILS) [33]. Lucene is a well-known open-source software that follows a classic IR-based approach to perform full-text search on text documents. On the other hand, WSQBE is an academic approach to Web Service discovery that combines classic IR techniques with a search-space reduction mechanism. Lastly, ILS is an acronym for another academic approach described in [33], since it combines classic IR techniques with term expansion based on lexical relations, and optionally compares the structure of services against a WSDL specification of the desired service, which the user who performs the discovery must supply.

The applied registries rank discovery results, in part, by the occurrences of terms that coexist in the query and published WSDL documents. Specifically, WSQBE gathers terms from names and comments of service port-types, operations, messages and definitions associated with exchanged data-types. Afterward, the gathered words are preprocessed for disambiguating naming conventions, removing stop-words and removing their endings (WSQBE employs Porter's stemming algorithm). WSQBE preprocesses queries to remove stop-words and endings before matching them onto available service descriptions. ILS not only preprocesses WSDL documents in a rather similar way, but also expands gathered terms using stems of the synonyms of the gathered terms, and stems of the words hierarchically related to the gathered terms, such as hypernyms, hyponyms and siblings.

On the other hand, out-of-the-box versions of Lucene do not preprocess documents, since the popular search engine has been designed for indexing any kind of textual documents [35]. Consequently, reserved words of WSDL, such as service, bindings or port, introduce noise to the search engine. Thus, we powered the Lucene-based registry with the preprocessing techniques proposed by WSQBE [19]. By doing so, we avoided introducing noise to the search engine, making it aware of WSDL and increasing its effectiveness to retrieve relevant services. From now on, we will refer to this version of Lucene as Lucene4WSDL.

With respect to the data-set, the body described in Section 5 was used to build 9 versions of the 391 WSDL documents (the original version plus eight new versions). Seven versions of the data-set were built by removing each anti-pattern individually. To do this, an experienced service-oriented application developer identified which anti-patterns were present in a given WSDL document. Then, he looked for their solutions in Section 4, and built as many versions of the given WSDL document as anti-patterns it had by separately removing the occurrences of each detected problem. Another version of the data-set was built by removing all the anti-patterns of each WSDL document.

It is worth noting that we omitted removing the Enclosed data model anti-pattern and building the corresponding data-set version. The reason behind this is that the same retrieval results are achieved by either enclosing data-set models or moving them to a separated schema file. This particular anti-pattern actually impacts on discovery when data models are designed to be reused and best practices for modelling data are followed, which requires to place these models on separated files. Nevertheless, this experiment measured to some extent the impact of these best practices since they are within the scope of other anti-patterns, like Ambiguous names or Redundant data models.

For the queries, we applied the 30 queries that are described in [19]. As the performance of the employed registries, and of any recommender system in general, depends on the data-set and the queries given as inputs, the same queries have been used with each registry. Each applied query consisted of relevant terms that were gathered from a Java interface describing the service functionality being discovered. Queries were preprocessed to remove Java reserved words and stop-words, and deal with JavaBeans naming convention. In addition, as the inquiry interface of ILS optionally accepts a functional description of the desired services using WSDL, we built a WSDL document for representing each query in the test data-set. To do this, we used a query processor that transforms a Java interface into a WSDL document by using the Java2WSDL[8] library, as explained in [19].

To measure the performance of the registries we used Recall-at-*n*, Precision-at-*n* and Normalized Recall. Recall-at-*n* computes how effective a discovery system is in retrieving relevant documents in a window of *n* documents [36]. Formally:

$$Recall\,at\,n = \frac{RetRel_n}{R}$$

Given a query with R relevant documents, Recall-at-*n* computes the number of relevant services up to n candidates in the result list, i.e. $RetRel_n$. Our goal is to achieve good Recall in a window of *only* 10 retrieved services, i.e. when $n = 10$. We have chosen this window size to obtain a good balance between the number of candidates and the number of relevant candidates retrieved. Moreover, we believe that a discoverer can easily examine 10 Web Service descriptions.

Precision-at-*n* computes precision at different cut-off points of the results list [36]. For example, if the top 10 documents are all relevant to a query and the next 10 are all non-relevant, we have a precision of 100% at a cut-off of 10 documents but a precision of 50% at a cut-off of 20 documents. Formally:

$$Precision\,at\,n = \frac{RetRel_n}{n}$$

We used two special cases of this metric, a case for $n = 1$ and another for $n = R$. The first case, Precision-at-*1*, indicates whether the first retrieved element is relevant. The second case computes the precision at the R^{th} position in the rank,

[8] Java2WSDL, http://ws.apache.org/axis/.

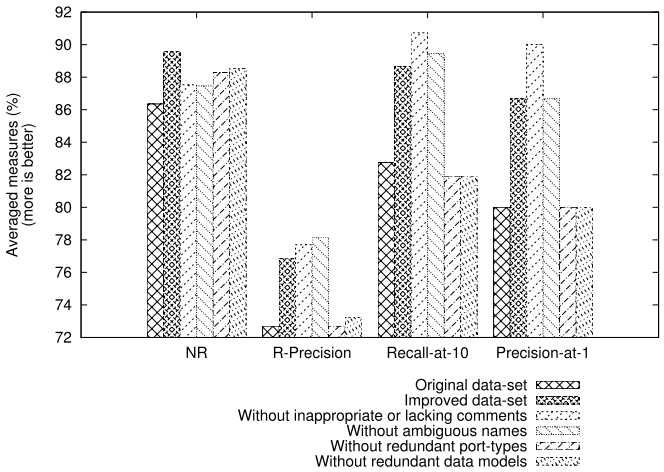


Fig. 7. Average measure results using Lucene4WSDL registry.

with R being the number of relevant documents for a given query. The Precision-at-R particular case is known as R-Precision; therefore, we will refer to Precision-at-R as R-Precision from now on. For example, if there are 10 documents relevant to the query within a data-set and they are retrieved before the 11^{th} position, we have an R-Precision of 100%, but if 5 of them are retrieved after the top 10 we have 50%.

The Normalized Recall (NR) measure takes into account Recall and the position of each relevant retrieved document within the result list [37]. NR is one of the most popular measures for evaluating and comparing information retrieval systems because it returns one single number in contrast to paired recall-precision measure. Formally, the NR of a query for a data-set of size N is:

$$NR = 1 - \frac{\sum_{i=1}^{R} r_i - \sum_{i=1}^{R} i}{R(N-R)}$$

We decided to employ these metrics because they not only characterize how well a search engine performs in finding relevant documents, or services in this case, but also they take into account the position of each relevant retrieved service within the result list. This fact makes these metrics especially suitable for comparing registries that retrieve services in a rank, like Lucene4WSDL, WSQBE and ILS do.

As some of these metrics require to know exactly the set of all services in the collection relevant to a given query (i.e. R) we have exhaustively analyzed the body of WSDL documents to determine the relevant services for each query. To do this, a developer judged whether the operations of a retrieved service fulfilled the expectations previously specified in each query. For example, if the developer required a Web Service for converting from Euros to Dollars, then a retrieved Web Service for converting from Yens to Dollars was considered non-relevant, even though these services were strongly related. In this particular case, only Web Services for converting from Euros to Dollars were relevant. It is worth noting that for any query there were, at most, 8 relevant services within the evaluation-set. Besides, there are 10 queries that had associated only one relevant service. This severely impacts on Precision-at-n measures since if the relevant service is not ranked first for these 10 queries, then the corresponding measures will be 0%. Note that these peculiarities make the validation of the discovery mechanism very strict.

We have calculated the aforementioned metrics for each query and each registry using the 9 versions of the data-set, and then averaged the results over the 30 queries per registry. The obtained results showed that separately remedying 3 of the identified anti-patterns, namely Whatever types, Undercover fault information within standard messages and Low cohesive operations of the same port-type, had an insignificant impact on the retrieval effectiveness of the employed registries. These results may stem from the fact that these anti-patterns are the least frequent among the analyzed WSDL documents. Specifically, the Whatever types anti-pattern affects 63 WSDL documents, the Undercover fault information within standard messages anti-pattern occurs in 59 documents, and the Low cohesive operations of the same port-type anti-pattern in 31 documents (see Fig. 4).

Figs. 7–9 show the average results of each metric when using the data-sets with Lucene4WSDL, WSQBE and ILS registries, respectively. In order to enable comparisons, we arranged the results in groups of six bars within each figure, in which each group is associated with an employed metric. From left to right, the first bar within each group represents the achieved results for each metric when using the original version of the data-set, the second bar represents the results when removing all anti-patterns from the data-set, the third bar represents the results when removing the Inappropriate or lacking comments anti-pattern, the fourth bar represents the results when removing the Ambiguous names anti-pattern, the fifth bar represents the results when removing the Redundant port-types anti-pattern, and the sixth bar represents the results when removing the Redundant data models anti-pattern.

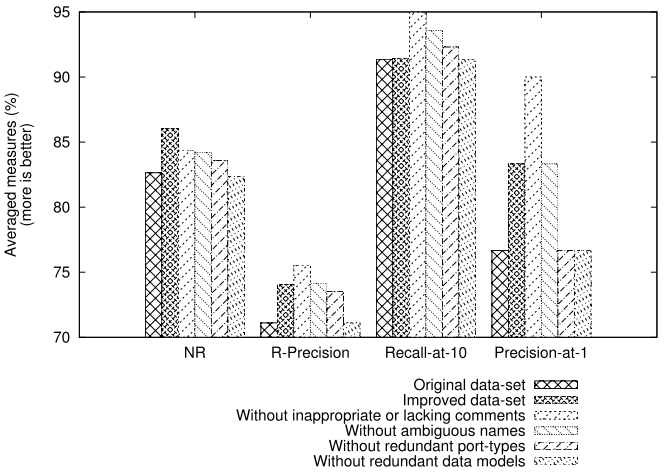


Fig. 8. Average measure results using WSQBE registry.

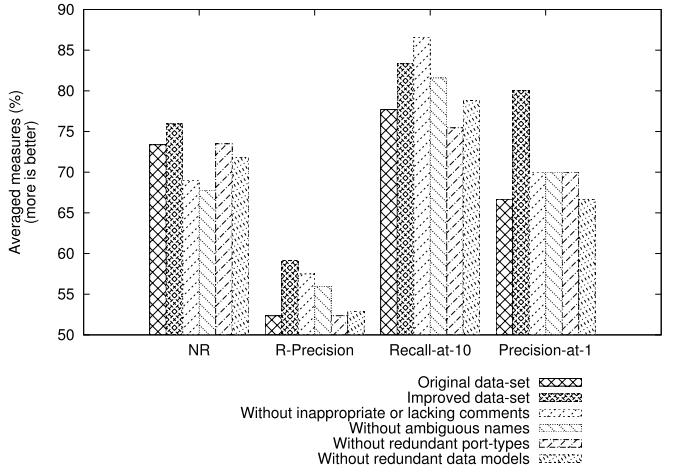


Fig. 9. Average measure results using ILS registry.

In general, Figs. 7–9 show that the evaluated registries achieved better performance measures when using any of the improved versions of the data-set than when using the original version. The biggest gain takes place in the results associated with the Precision-at-*1* measure. The experiments when removing the occurrences of the Inappropriate or lacking comments anti-pattern showed that the Lucene4WSDL, WSQBE and ILS registries improved their Precision-at-*1* by 10%, 13.33%, 3.33%, respectively. Figs. 7–9 show that the tendency to improve Precision-at-*1* results was maintained by the removal of the other anti-patterns as well. Having a higher Precision-at-*1* means that the registry performs better in retrieving a relevant service at the top of the result list. As supported by different experiments, these results have a great impact on discoverability because users tend to select higher ranked search results [38]. For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the second ranked one is, at most, 60% [38].

The *R*-Precision results have provided more evidence of the improvements in the retrieval of relevant services before non-relevant ones when removing anti-patterns from the data-set. In addition, the Recall-at-*10* results have empirically shown that the employed registries were more effective in retrieving more relevant services when using the improved data-sets. Moreover, when removing all the anti-patterns the employed registries have empirically shown to improve Normalized Recall. As this measure takes into account Recall and the position of each relevant retrieved service within the result list, the results confirmed that the employed registries not only retrieved more relevant services, but also ranked them first in the result list when using the WSDL documents without any anti-pattern.

These positive results may stem from the fact that the original WSDL documents usually contain redundant, meaningless and nonspecific terms; i.e., when there is no a strong connection between the operation functionality and the terms used to define it (e.g., "calculate(int i0, int i1):int"). For example, the term "parameters" is frequently used for naming inputs and "return" for outputs, whereas "body" is usually associated with operations bound to HTTP protocol. Table 2 lists the part names most frequently encountered within the original version of the studied data-set.

Table 2
Commonest message part names.

Name	Occurrences	Frequency
Parameters	2124	16.32
Body	2044	15.57
Return	524	3.99
Password	473	3.6
Userid	275	2.09

Specifically, the original WSDL documents have 3368 unique terms, but after removing the identified anti-patterns they only have 2555 unique terms. Indeed, as the proposed anti-pattern solutions remove meaningless nonspecific terms and add representative names, the refactored WSDL documents have less terms but they are more representative of the functionality of the services. This kind of terms strongly impacts the retrieval performance of syntactic approaches to service discovery [18–20,27]. Therefore, we could expect retrieving a specific relevant service near to the top of the list when specific and representative service descriptions have been published in the registry, which is the case when using the improved WSDL documents.

7.2. Evaluation of users' ability to understand WSDL documents

We conducted an experiment with software engineering students and professionals and asked them about the intelligibility of different versions of WSDL documents. Concretely, the participants were given several WSDL documents and a questionnaire designed to analyze the implications of those anti-patterns that occur less frequently among the WSDL documents of the analyzed data-set, namely Low cohesive operations of the same port-type, Undercover fault information within standard messages, Whatever types and Redundant data models.

To build the employed WSDL documents, we took service descriptions that have the aforementioned 4 anti-patterns from the data-set used for the discovery experiment. Then, for each service we generated an improved WSDL document by removing the anti-pattern occurrences. The participants answered several questions about the readability of original version of each WSDL document. Afterward, they were asked about the improved versions. The questionnaire was given as a homework to 26 participants who were taking a SOC course, and the data were collected on the 18^{th} of September. The group of participants was integrated by last year software engineering students and practicing software engineers. It is worth noting that the participants did not know about the catalog of discoverability anti-patterns proposed in this paper or its related works, until the survey was finished.

The questionnaire consisted of eleven questions divided into three groups. A group of questions were designed to familiarize the participants with each version of the employed WSDL documents. For example, a question asked about the number of operations offered by a service. Another group of questions asked the participants about whether the WSDL documents were explanatory enough to they understand what the offered service does and how to use it, or if their descriptiveness could be improved to some extent. The last group of questions allowed participants to comment which version of the employed WSDL documents would outsource and why. The questions of the second and third groups, and the main results of the survey are described next.

First, we gave the participants a service port-type with several operations belonging to the same domain, but one operation of a different domain, and in turn asked the participants whether removing the non-cohesive operation would improve the understandability of the service or not. In other words, we were implicitly asking individuals if they conceived Low cohesive operations of the same port-type anti-pattern as a problem that would hinder understanding the service, and if they would apply its associated remedy. The results showed that 92% of the participants implicitly answered that they would remove the anti-pattern.

Second, we gave the participants a service operation that returns a data-type based on the xsd:any constructor, and whose documentation provides hints that, in case of an execution problem, error information would be included in the output message. Then, we asked the participants three questions. The first question was about whether they could determine the structure of the operation response. The second one asked them about whether they would replace the data-type of the operation output with a data-type that merely represents the operation result. Finally, the third question was if they preferred piggybacking error information in output messages or exchanging it in fault messages. In other words, we implicitly asked the participants to evaluate whether the Whatever types and Undercover fault information within standard messages anti-patterns took place in the analyzed WSDL document, and whether they would apply the proposed remedies or not.

As a result, 92% of the participants answered that the structure of the output could not be determined. The rest of the participants answered that the analyzed operation always returns instances of xsd:double or xsd:int data-types. This result may stem from the fact that the operation is for uploading files, and if a file is successfully transferred, then the file size is returned. In this sense, it seems that 8% of the participants disregarded the possibility of a failure during the execution of the

operation. The results also showed that 92% of the participants would replace the data-type of the output of the operation. As the reader can see, the percentage of participants that identified the Whatever type problem was exactly the same that voted for replacing the data-type definition.

On the other hand, 92% of the participants realized that the analyzed output message could exchange error information. However, 81% of the them answered that they would use fault message to convey error information. This may be related to the fact that fault messages are curiously uncommon among the WSDL documents of the data-set.

In order to collect opinions about the Redundant data models anti-pattern, we gave to the participants a WSDL document with two operations returning the same data-types, but defined twice. The participants were asked whether they would remove one of the redundant data-types or not. Eighty one percentof them answered that they would remove the repeated data-types.

For the next group of questions, we gave the participants the improved version of each WSDL document that they had analyzed. Finally, we asked the participants about which version of the analyzed WSDL documents they would outsource. Additionally, each participant gave his/her opinions about why they would select one or another version. The results showed that 84% of the participants would use the improved WSDL document, 8% the original version and 8% any version indistinctly.

The comments made by the participants provided an idea of the reasons behind choosing the improved versions. Some participants included two, or more, different reasons in their comments. From these comments we summarized and ranked the most frequent main reasons. Accordingly, the reasons are listed in decreasing order of occurrence next:

1. the data-types exchanged by the improved WSDL document were better represented,
2. the improved WSDL document was more concise,
3. the operations of the improved WSDL document belonged to a single domain,
4. error information was better handled by the improved WSDL document.

Specifically, the results showed that 16 participants included in their comments the reason related to exchanged data-type definitions. The responses of 13 participants highlighted that the improved WSDL documents were more concise than the original versions. Eight participants commented that they would outsource the improved WSDL documents because they arranged cohesive operations. Finally, 6 participants said that they would outsource the improved versions because separating error information from output messages helped them to understand how to access the service.

As the reader can see, the reasons given by the participants express the results of remedying some of the identified anti-patterns. In particular, improving the XSDs of the data-types exchanged by the analyzed services, which was ranked first, was caused by remedying the Redundant data models and Enclosed data model anti-patterns. These anti-patterns are also associated with the second reason of the rank, which takes under consideration the length of the analyzed WSDL documents, given that remedying both aforementioned anti-patterns causes conciser WSDL documents. The other two reasons given by the participants are related to remedying the Low cohesive operations in the same port-type and Undercover fault information within standard messages anti-patterns, respectively.

8. Future work

Future work related to the identified anti-patterns is planned in two directions. The first direction is mainly experimental, and is aimed to collect more evidence of the anti-patterns impact on discovery using a recently published repository of real Web Services.[9] The other direction is concerned with algorithms for automatically detecting and remedying Web Service discoverability anti-patterns.

As explained before, detecting an anti-pattern is strongly related to the way it manifests itself. With regard to the detection of Evident anti-patterns, this requires one to analyze whether the anti-patterns of this category are present in the syntax of WSDL documents. We have defined, implemented and evaluated several syntax-based algorithms. Preliminary results measuring the precision of these algorithms are encouraging. On the other hand, the detection of Not immediately apparent anti-patterns presents several challenges.

For the detection of the Low cohesive operations in the same port-type anti-pattern, we are researching on a heuristic based on machine learning classification algorithms. The idea is to use an automatic classifier to deduce the domain of each operation within a port-type and, in turn, check whether deduced domains are similar. One of the difficulties that we have encountered is that the accuracy of such a classifier is improved if it is trained previously. This implies that the heuristic needs as input a collection of WSDL documents, whose operations have been properly classified.

Regarding the detection of Ambiguous names and Inappropriate comments anti-patterns, we are evaluating an heuristic that combines an electronic lexical database [39] and a natural language parser [40]. Our goal is to analyze whether WSDL documents follow the conventions described in Sections 4.1 and 4.2, i.e. part names length should be between 3 and 15 characters, operation names should be in the form: <verb> "+" <noun>, composed names should be written according to common notations, etc.

Finally, with respect to the Undercover fault information within standard messages anti-pattern, we have implemented an algorithm to check whether the definition of an output parameter is according to a general purpose data-type or a "three-parts" approach. As explained in Section 4.8, normally this anti-pattern takes place when an output parameter follows such

[9] The QWS Dataset, http://www.uoguelph.ca/~qmahmoud/qws/index.html.

kind of definitions. One of the problems that we have encountered is that in many cases is still necessary to invoke an operation and analyze its results, in order to actually determine the presence of the anti-pattern.

All in all, the automatic detection of several of the identified anti-patterns is still an open problem. We expect that the aforementioned algorithms and heuristics will provide assistance to developers for coding WSDL documents. In the end, the idea is not only assisting developers in making more representative WSDL documents by identifying anti-patterns, but also suggesting suitable refactoring operations.

9. Conclusions

In this article we have studied the implications of bad practices in the creation of WSDL documents on their understandability and discoverability when using syntactic approaches. We have analyzed a body of 391 real Web Services, and found eight common practices that may hinder human discoverers from understanding the services, and degrade the retrieval effectiveness of syntactic registries. In addition, we have supplied each identified bad practice with a reproducible solution. Therefore, this paper presents a novel catalog of eight WSDL discoverability anti-patterns.

We have exhaustively revised each WSDL document of a public data-set to analyze the occurrences of anti-patterns. As a result, we have found that 82% of the documents suffer from at least one anti-pattern. Afterwards, we have assessed the retrieval effectiveness of three discovery mechanisms when using the original WSDL documents and the improved ones; i.e., the WSDL documents that have been refactored according to the proposed solution of each anti-pattern. The fact that the results related to the improved data-sets surpass those achieved by using the original data-set regardless the approaches to service discovery employed, provides empirical evidence that suggests that the improvements are explained by the removal of discoverability anti-patterns rather than the incidence of the underlying discovery mechanism.

We also performed a survey to collect subjective opinions from software engineering students and practicing engineers about how intelligible several WSDL documents were. The results of the survey suggest that WSDL documents should be improved to increase service chances of being understood and, in turn, outsourced.

The experiment also shows that, an experienced service-oriented application developer requires 15 min on average to enhance a WSDL document, and verify that its semantics have not changed. Therefore, we conclude that manually enhancing WSDL documents should be incorporated as a development task because 15 min is a reasonable time investment with a favorable outcome of making services easier to be understood, and discovered by potential consumers.

The combination of the experiment related to the retrieval process of the service registries, and the questionnaire was designed to consider discovery from the point of view of the algorithms that support it as well as human developers, who have the final word on which service is more appropriate. Note that these results can vary with other data-sets and participants, as retrieval effectiveness improvements can be data-set specific and answering the survey depends on personal judgment. Nevertheless, as our approach relies on removing meaningless or unnecessary information and incorporating self-explanatory names and comments, it is reasonable to expect at least a small improvement on the discoverability and intelligibility of services when removing WSDL anti-patterns, versus preserving them.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper. We also thank Cristian Mateos for helping us to perform the survey. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

[1] C.W. Krueger, Software reuse, ACM Computing Surveys 24 (2) (1992) 131–183.
[2] M. Huhns, M. Singh, Service-oriented computing: Key concepts and principles, IEEE Internet Computing 9 (1) (2005) 75–81.
[3] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology, and Design, Wiley, 2005.
[4] M. Bichler, K.-J. Lin, Service-oriented computing, Computer 39 (3) (2006) 99–101.
[5] S.J. Vaughan-Nichols, Web Services: Beyond the hype, Computer 35 (2) (2002) 18–21.
[6] R. McCool, Rethinking the semantic Web. Part I, IEEE Internet Computing 9 (6) (2005) 86–88.
[7] H. Wang, J. Huang, Y. Qu, J. Xie, Web Services: problems and future directions, Journal of Web Semantics 1 (3) (2004) 309–320.
[8] J. Garofalakis, Y. Panagis, E. Sakkopoulos, A. Tsakalidis, Contemporary Web Service discovery mechanisms, Journal of Web Engineering 5 (3) (2006) 265–290.
[9] M.B. Blake, M.F. Nowlan, Taming Web Services from the wild, IEEE Internet Computing 12 (5) (2008) 62–69.
[10] J. Fan, S. Kambhampati, A snapshot of public Web Services, SIGMOD Record 34 (1) (2005) 24–32.
[11] M. Sabou, J. Pan, Towards semantically enhanced Web Service repositories, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 142–150.
[12] A. Gomez-Perez, O. Corcho-Garcia, M. Fernandez-Lopez, Ontological Engineering, Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 2003.
[13] M. Shamsfard, A. A. Barforoush, Learning ontologies from natural language texts, International Journal of Human–Computer Studies 60 (2004) 17–63.
[14] R. McCool, Rethinking the Semantic Web. Part II, IEEE Internet Computing 10 (1) (2006) 93–96.
[15] H. Song, D. Cheng, A. Messer, S. Kalasapur, Web Service discovery using general-purpose search engines, in: IEEE International Conference on Web Services (ICWS), 2007, pp. 265–271.
[16] E. Al-Masri, Q. Mahmoud, Discovering Web Services in search engines, IEEE Internet Computing 12 (3) (2008) 74–77.
[17] M. Paolucci, K. Sycara, Autonomous semantic Web Services, IEEE Internet Computing 7 (5) (2003) 34–41.

[18] X. Dong, A.Y. Halevy, J. Madhavan, E. Nemes, J. Zhang, Similarity search for Web Services, in: M.A. Nascimento, M.T. Özsu, D. Kossmann, R.J. Miller, J.A. Blakeley, K.B. Schiefer (Eds.), (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Morgan Kaufmann, Toronto, Canada, 2004, pp. 372–383.
[19] M. Crasso, A. Zunino, M. Campo, Easy Web Service discovery: A Query-by-Example approach, Science of Computer Programming 71 (2) (2008) 144–164.
[20] N. Kokash, W.-J. van den Heuvel, V. D'Andrea, Leveraging Web Services discovery with customizable hybrid matching, in: A. Dan, W. Lamersdorf (Eds.), International Conference on Service-Oriented Computing, in: Lecture Notes in Computer Science, vol. 4294, Springer Verlag, Chicago, IL, USA, 2006, pp. 522–528.
[21] M. Henning, API design matters, Communications of ACM 52 (5) (2009) 46–56.
[22] W.J. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley, 1998.
[23] J. Pasley, Avoid XML schema wildcards for Web Service interfaces, IEEE Internet Computing 10 (3) (2006) 72–79.
[24] J. Beaton, S.Y. Jeong, Y. Xie, J. Jack, B.A. Myers, Usability challenges for enterprise service-oriented architecture APIs, in: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2008, pp. 193–196.
[25] P.C. Pendharkar, J.A. Rodger, An empirical study of factors impacting the size of object-oriented component code documentation, in: SIGDOC '02: Proceedings of the 20th Annual International Conference on Computer Documentation, ACM Press, New York, NY, USA, 2002, pp. 152–156.
[26] N. Kokash, A comparison of Web Service interface similarity measures, in: 3rd European Starting AI Researcher Symposium, IOS Press, Riva del Garda, Italy, 2006, pp. 220–231. (Local-chair Loris Penserini).
[27] W. Jian, W. Zhaohui, Similarity-based Web Service matchmaking, in: IEEE International Conference on Services Computing, vol. 1, IEEE Computer Society, Orlando, Florida, USA, 2005, pp. 287–294.
[28] Y. Wang, E. Stroulia, Flexible interface matching for Web Service discovery, in: WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering, IEEE Computer Society, Washington, DC, USA, 2003, p. 147.
[29] A. Akram, R. Allan, D. Meredith, Best practices in Web Service style, data binding and validation for use in data-centric scientific applications, in: Proceedings of UK e-Science All Hands Conference, Imperial College of London, 2006.
[30] D. Kramer, Api documentation from source code comments: A case study of javadoc, in: SIGDOC '99: Proceedings of the 17th Annual International Conference on Computer documentation, ACM, New York, NY, USA, 1999, pp. 147–153.
[31] W3C Consortium, Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Candidate Recommendation. http://www.w3.org/TR/wsdl20 (Jun. 2007).
[32] E. Yourdon, L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1979.
[33] E. Stroulia, Y. Wang, Structural and semantic matching for assessing Web Service similarity, International Journal of Cooperative Information Systems 14 (4) (2005) 407–438.
[34] A. Heß, E. Johnston, N. Kushmerick, ASSAM: A tool for semi-automatically annotating semantic Web Services, in: S.A. McIlraith, D. Plexousakis, F. van Harmelen (Eds.), International Semantic Web Conference, in: Lecture Notes in Computer Science (LNCS), vol. 3298, Springer, Hiroshima, Japan, 2004, pp. 320–334.
[35] E. Hatcher, O. Gospodnetic, Lucene in Action (In Action series), Manning Publications Co, 2004.
[36] R.R. Korfhage, Information Storage and Retrieval, John Wiley & Sons, 1997.
[37] P. Bollmann, The normalized recall and related measures, in: Proceedings of the 6th annual international ACM SIGIR conference on Research and development in information retrieval, 1983, pp. 122–128.
[38] E. Agichtein, E. Brill, S. Dumais, R. Ragno, Learning user interaction models for predicting web search result preferences, in: 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06), Seattle, Washington, USA, ACM Press, New York, NY, USA, 2006, pp. 3–10.
[39] C. Fellbaum, WordNet: An Electronic Lexical Database, in: Bradford Books, 1989.
[40] D. Klein, C.D. Manning, Accurate unlexicalized parsing, in: ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, Morristown, NJ, USA, 2003, pp. 423–430.